Fermi National Accelerator Laboratory

TM-1679

CANVAS: C++ Objects for Easy Graphics on an Evans & Sutherland PS390 Terminal. User's Guide

Leo Michelotti and Richard Kick
Fermi National Accelerator Laboratory
P.O. Box 500
Batavia, Illinois 60510

August 27,1990

Operated by Universities Research Association Inc. under contract with the United States Department of Energy

CANVAS: C++ objects for easy graphics on an Evans & Sutherland PS390 terminal. User's Guide

Leo Michelotti Richard Kick

August 27, 1990

1 Introduction

The Evans and Sutherland PS390 terminal which resides in the X-Gallery HiBay area is a powerful, sophisticated graphics engine whose price/performance ratio approaches infinity because practically no one uses it.[1] The reason for this lack of enthusiasm is obvious: no self-respecting scientist still of sound mind would willingly choose to read, much less assimilate, the seven-volume set of manuals that accompanies this device. The information buried in those volumes is, of course, essential for programming the PS390 to do all the sophisticated things it is capable of doing, such as creating automobiles with doors that open and close and steering wheels which move synchronously with the front tires, all at the turn of a knob. Regrettably, doing simple things, such as viewing three-dimensional magnetic fields, is no easier than doing the most complicated ones.[2] There is at least one brand of third party software available for viewing data, but it requires specially formatted data files, it does not allow for data interaction, and as is usually the case with self-contained programs, it invariably does not do the one or two tasks needed for a given application. Another approach was used by Fermilab's Computing Division in writing a DI-3000 device driver for the PS390 terminal. This was a valid and useful project, but it must have been like trying to fit a square peg into a round hole. The PS390 is Phigs-like in its graphics protocols while DI-3000 is based on ACM Core, which was devised when graphics was viewed primarily as a means of drawing pictures rather than interacting with data.

The C++ classes described in this note comprise an attempt to provide an object-oriented approach, and if there was ever a graphics terminal naturally suited to object-oriented programming, the PS390 is it. Since a canvas is not a program but *a variable to be used in programs*, users can write software to suit their particular needs. By simply declaring canvas variables the application program is provided with an object which accepts data and displays it automatically. Any number of canvases can be placed anywhere on the screen, so data can be viewed in a variety of ways simultaneously. Further, the "real-time" transformation

[1] An excellent Macintosh commercial was developed along these lines. Two executives are discussing how to evaluate hardware performance. One suggests that the answer lies in counting "megaflops or megahertz something like that." The other counters that an easier way is to observe which computer people use most frequently. When the first objects that the criterion is not fair because people *like* to use a Macintosh he receives a withering look, suggestive of severe reductions in prospects for future promotion.

[2] Perhaps this is just the negative way of saying that complicated graphics is no harder than simple graphics.

capabilities of the PS390 are activated in one step by "connecting" its external devices, the dials and the puck, to the desired canvas. There is no need for the applications programmer to construct his own function networks, choose names for nodes, and do any of the other administrative tasks laid out in the manuals, including connecting the terminal to a host computer and initializing it. These are handled automatically by the canvases themselves, thus removing this clutter from the application program.

This User's Guide is arranged as follows. We offer in Section 2 four sample programs which illustrate the simplest usage of the canvas variables: **canvas3d**, **canvas2d**, and **canvasRas**. This is follwed by a description of (currently) available methods in Sections 3 and 4, the former dealing with object-orinted graphics and the latter with raster graphics. What will be described is a minimal, "bare bones" implementation which we hope will be useful to anyone who wants an easy way to explore three-dimensional data without frills. It is not our objective to attempt, single-handedly, to provide an Interviews package for the PS390.[3] However, features will continue to be added slowly as needed, especially with regard to (a) locate, pick, and identify functions, (b) three-dimensional cursors, (c) ready-to-use graphics objects, such as **beamLine** or **surface**, (d) new types of derived canvases, such as **stripChart**, **menu**, or **tuneDiagram**, and (e) construction of a **canvas4d** class. For those who might want to use these variables and their methods libraries and documentation can be found in the area `ALMOND::USR$DISK4:[MICHELOTT.PUBLIC.CANVAS]`.

2 Sample programs

The first two programs in this section were suggested by colleagues in Accelerator Division; the other two had no motivation other than to illustrate the usage of canvas variables. Almost complete listings are provided, but for our purposes, it is certainly not necessary to understand all their details. We shall walk through the important highlights of each.

All of these sample programs are available to anyone who might want to play with them to see what they actually look like on the screen. Simply (i) walk up to the XGal Hi Bay Mezzanine and log into your ALMOND account using the PS390 terminal; (ii) browse through the demo names by doing

```
$ DIR ALMOND::USR$DISK4:[MICHELOTT.PUBLIC.CANVAS.DEMOS]*.DEMO;
```

and (iii) invoke a demo with

```
$ @ALMOND::USR$DISK4:[MICHELOTT.PUBLIC.CANVAS.DEMOS]<name>.DEMO
```

where < *name* > is the name of one of the demos seen in Step (ii). Further, it is possible to make hardcopies of the PS390 display (with the exception of **canvasRas** images) by entering

```
$ @ALMOND::USR$DISK4:[MICHELOTT.PUBLIC.CANVAS]HARDCOPY
```

at VMS system level. A PostScript file named PS800.DAT will be written into your default directory.

2.1 Data from files

This first program merely reads three-dimensional data from a set of files and displays them in a canvas.[4] Each file contained one-day temperature profiles at one of twenty-four stations around the ring. Each line

[3] Incidentally, the fact that our class has the same name as that used by SUN graphics is purely a coincidence and is not meant to suggest any correspondence.

[4] The program in this section was suggested by Walter Kissel.

contained a (*time*, *temperature*, *station*) triplet, and it was desired to display these as twenty-four different-colored curves. The *names* of all the data files were written into a separate file for program access.

We shall mark the important features of this program, flagged by their line numbers:

line 6: The canvas resource file, **canvas.rsc**, must be included before any canvas variables are declared and must have file scope, that is, it must be included *outside* of all "{ ...}" brackets (the most logical place being near the top of the file, as shown here). This header can be read from the area `ALMOND::USR\$DISK4:[MICHELOTT.PUBLIC.HEADERS]`.

line 11: A number of forms are available for declaring canvas variables. This one employs two character-string arguments to instance **pic**, a three-dimensional canvas. The first argument specifies how graphics transformations are to be carried out; we shall put off its detailed explanation to Section 3.1. The second requests a parallel, rather than perspective, projection from the three-dimensional "world" of data into the screen; perspective projections are the default.

lines 13-20: This fragment of code serves the dual purpose of placing axes within the canvas and alerting it to the data ranges it can expect. Each data file contains a 24 hour temperature profile at one of the the 24 stations around the ring. We choose temperature to be the z-axis, while "hour" and "station" will be the other two. The array `minmax` indicate the minimum and maximum values associated with each axis, while `delta` specifies the spacing between tic marks.

lines 22-25: Anticipating the desire to look at the data from various views, the PS390 dialbox is connected to the canvas's transformation "nodes." There are three basic data transformations which can be carried out: rotations, dilations, and translations. These statements will allow the first three dials to control rotations in three independent directions, the fourth, dilation (which is uniform in all three directions), and the last three, translations. Dial 5 is not connected to anything.

lines 8, 33: Names of the data files are contained in a separate file, whose name will appear in the command line invoking the program, as in

```
$ kissel namefile.dat
```

lines 37-54: These two `while` loops form the core of the program. The first reads the names of the data files, the second, the data. The temperature profiles are modelled as wireframes with the functions **.startFrame** (line 45), which initializes a wireframe, and **.addToFrame** (line 49), which continues it. Since transformations have already been connected, the dialbox can be used to manipulate the image *as it is being drawn*.

line 53: It frequently happens at the end of a program that the last few data points sent to a canvas do not reach it because the Ethernet buffers have not been filled. The method **.purgeBuff** flushes these buffers and assures that all the data are displayed.

lines 58-61: Finally, we delay leaving the program. Upon exiting, canvas destructors are invoked, and they reset the PS390, destroying all the objects which have been created. Without the delay, this would happen before the user had a chance to examine his data. If it is desired to exit a program *while* leaving the objects intact – to make a hard copy of the screen, for example – then exit disgracefully with a `^Y`.

PROGRAM LISTING

```
1     #include <stddef.h>     //  These two statements sideline
2     #include <errno.h>      //  annoying compiler warnings
3                             //  about PSECT ERRNO
```

```
4     #include <stdio.hxx>
5
6     #include "canvas.rsc"
7
8     main ( int argNum, char *fileList[] ) {
9
10    // Declaring the canvas variable ..............
11    canvas3d pic ( "OOW", "orthographic" );
12
13    // Setting up the axis system .................
14    double  delta[3];
15    double minmax[6];
16    //  ____ x _______   ________ y ______      ______ z _______
17     minmax[0]  =   1.0;  minmax[2] =  1.0;    minmax[4] =  60.0;
18     minmax[1]  =  24.0;  minmax[3] = 24.0;    minmax[5] = 120.0;
19      delta[0]  =   1.0;   delta[1] =  1.0;     delta[2] =  10.0;
20    pic.makeAxes( minmax, delta );
21
22    // Connect transformations to dials ............
23    pic.connRot   ( 1, 2, 3 );
24    pic.connScale ( 4 );
25    pic.connTrans ( 6, 7, 8 );
26
27    // Finally, read and display the data ..........
28    FILE *dataFile;
29    FILE *nameFile;
30    char nameOfDataFile[ 80 ];
31
32
33    nameFile = fopen( fileList[1], "r" );
34
35    char firstPoint = 1;
36    double time, temperature, station;
37    while ( fscanf( nameFile, "%s", nameOfDataFile ) != EOF ) {
38      dataFile = fopen( nameOfDataFile, "r" );
39      while ( fscanf( dataFile,    "%le  %le  %le",
40                      &time, &temperature, &station
41                    )
42            != EOF
43            ) {
44        if ( firstPoint ) {
45          pic.startFrame( station, time, temperature );
46          firstPoint = 0;
47          continue;
48          }
49        pic.addToFrame( station, time, temperature );
50        }
```

```
51      fclose( dataFile );
52      firstPoint = 1;
53      pic.purgeBuff();
54      }
55
56    fclose( nameFile );
57
58    // Delaying program exit .....................
59    char  dummyString[ 80 ];
60    printf("\nPress RETURN to exit the program.\n");
61    gets( dummyString );
62
63    }
```

2.2 Eyeball curve-fitting

It frequently happens that one would like to fit to data from a parametrized family of curves, not by using automated (least squared error) methods but simply by looking at the curves and playing with the parameters until a decent fit is achieved *at localized regions of interest*. The program listed below enables this type of "exploratory data analysis."[5]

lines 16-17, 146-157: This program had an interesting problem associated with it. I wanted two two-dimensional canvas variables with file scope on the screen simultaneously. However, the declarative forms employed (lines 151 and 156) could be used only from within a function. The solution was to declare pointers to canvases (lines 16 and 17) which are assigned values *after* the canvases are declared (lines 152 and 157). This illustrates strongly that a C++ class acts *completely* like the original variables of the language.

Locations of the canvases on the screen are specified by two arrays, `rt` and `lb`, which contain "normalized screen coordinates" - lying within ± 1 - of the right-top and left-bottom corners. Thus the canvas `fit` is to be centered vertically and occupy the left half of the screen, while `resid` will occupy the right half. (Compare lines 151 and 156 with line 11 of the preceding section.)

lines 159-165: After the canvases have been declared, translation nodes are connected to dials, as before. We are not interested in rotations for this application.

lines 172-176: The file containing (x, y) coordinate pairs is read and the data stored in two arrays. Line 176 contains one of those multi-purpose statements that non-C programmers sometimes find confusing. It simultaneously (a) reads the data file, storing one pair at a time, (b) counts the number of pairs stored, and (c) terminates when the end of file is reached.

lines 178-211: Then comes a block of code which makes an initial estimate for the parameters of a gaussian model from the data's statistics. Since it is not of interest, in the context of this document, we have omitted it.

lines 213-224: As in the previous example, axes are established, one for each canvas. The vertical axis parameters are different for the two windows, since `fit` is to display the data themselves, while `resid` will contain the display of residuals from the gaussian model.

[5] The program in this section was suggested by Craig Moore.

lines 227-240: Three objects are to be plotted in the two canvases: the scatterplot of (x, y) coordinate pairs, the gaussian curve through those points, and the set of residuals. These are respectively given the tokens, already declared as **char*** variables in lines 13-15, of **dataGraf**, **curve**, and **residuals**. **dataGraf** is plotted using the methods **.startPts**, to initialize, and **.addToPts**, to complete the display (lines 228-229). The methods **.startFrame** and **.addToFrame** implement the display of **curve** and **residuals** as wireframes. The colors are chosen from a template made available by **canvas.rsc**: `BLUE`, `MAGENTA`, `RED`, `YELLOW`, `GREEN` and `CYAN`. Other colors can be obtained by linear interpolation, as in "`0.3*RED + 0.7*YELLOW`." The loop beginning at line 235 finishes the wireframes. Note that in addition to **.addToFrame**, the plot of residuals makes use of a method **.joinToFrame** (line 237). This enables **residuals** to be constructed from disconnected pieces, while **curve** is a completely connected wireframe.

lines 8, 242-249: The C++ class **ttMenu** was written some time ago and is convenient for writing programs with interactive steps. Its usage should be obvious from the context.

lines 64-141: The functions **absChange** (lines 64ff) and **relChange** (lines 103ff) simply allow the user to alter the parameters of the gaussian either directly or by choosing multipliers or summands. The details are not of interest, *except for* lines 98 and 138: after changes are made, the two objects **curve** and **residuals** are redrawn ...

lines 49-62: ...using the function **reDraw()**. After the old versions are erased from the canvases using **.remove** (lines 51-52), the new ones are put in their places: lines 53-61 almost exactly duplicate lines 232-241, in which these objects were originally drawn. The principal difference is that **reDraw** uses the pointers **fitPtr** and **residPtr** to refer to canvases.

lines 22-41: This block of code simply contains the gaussian function which is to be used as the family of curves for the fitting. The user will be able to manipulate four parameters (lines 27-30) in order to achieve a "good" fit.

PROGRAM LISTING

```
1     #include <stddef.h>
2     #include <errno.h>
3     #include <stdio.hxx>
4     #include <stream.hxx>
5     #include <math.hxx>
6
7     #include "canvas.rsc"
8     #include "[michelott.cpp.headers]menu.hxx"
9
10    #define       MAXDATA   256
11    const double  twoPi =     6.2831853072;
12
13    char*     dataGraf;
14    char*     curve;
15    char*     residuals;
16    canvas2d* fitPtr;
17    canvas2d* residPtr;
18    double    x[MAXDATA], y[MAXDATA];
19    int       num;
20
```

```
21
22    // ****************************
23    //
24    // Gaussian distribution to be used in fitting
25    //
26
27    double gsAverage       = 0.0;
28    double gsSigma         = 1.0;
29    double gsNormalization = 1.0;
30    double gsPlatform      = 0.0;
31
32    double gaussian ( double u ) {
33      double w;
34      double arg;
35      arg = ( u - gsAverage ) / gsSigma;
36      arg = - 0.5 * arg * arg;
37      w = exp( arg ) / ( sqrt( twoPi ) * gsSigma );
38      w *= gsNormalization;
39      w += gsPlatform;
40      return w;
41      }
42
43
44    // **************
45    //
46    // Manipulation routines in mainMenu
47    //
48
49    void reDraw() {
50    int i;
51    fitPtr   -> remove( curve );
52    residPtr -> remove( residuals );
53    curve     =   fitPtr -> startFrame( x[0], gaussian( x[0] ), RED    );
54    residuals = residPtr -> startFrame( x[0], 0.0,              YELLOW );
55                residPtr -> addToFrame( x[0], y[0] - gaussian( x[0] )  );
56    for ( i = 1; i < num; i++ ) {
57      fitPtr   -> addToFrame  ( x[i], gaussian( x[i] )         );
58      residPtr -> joinToFrame ( x[i], 0.0                      );
59      residPtr -> addToFrame  ( x[i], y[i] - gaussian( x[i] )  );
60      }
61    fitPtr->purgeBuff();
62    }
63
64    void absChange() {
65    int choice;
66    double value;
67
```

```
68    start:

........ <<< Lines 69-93 omitted >>> ..........

94      default:
95        break;
96      }
97
98    reDraw();
99    goto start;
100
101   }
102
103   void relChange() {
104   int choice;

........ <<< Lines 105-136 omitted >>> ..........

137
138   reDraw();
139   goto start;
140
141   }
142
143
144   main ( int argNum, char *fileName[] ) {
145
146   // Declaring the canvas variables ..............
147   double  rt[2], lb[2];
148   //   --------x   ---------------y
149   lb[0]  =  -1.0;  lb[1]  =  - 0.5;
150   rt[0]  =   0.0;  rt[1]  =    0.5;
151   canvas2d fit   ( rt, lb, "WWW" );
152   fitPtr = &fit;
153   //  ............................
154   lb[0]  =   0.0;  lb[1]  =  - 0.5;
155   rt[0]  =   1.0;  rt[1]  =    0.5;
156   canvas2d resid ( rt, lb, "WWW" );
157   residPtr = &resid;
158
159   // Connect transformations to dials ............
160   resid.connScale ( 1, "Scale" );
161   resid.connTrans ( 2, "Horiz",
162                     3, "Vert" );
163   fit.connScale   ( 5, "Dilation" );
164   fit.connTrans   ( 6, "Horiz",
165                     7, "Vert" );
```

```
166
167   // Reading data from the file .....
168
169   FILE*   xyFile;
170   int     i;
171
172   xyFile = fopen( fileName[1], "r" );
173
174   // Read the data file
175   num = 0;
176   while( fscanf( xyFile, "%lf  %lf", x + num, y + num ) != EOF )  num++;
177
178   // Find data scales
179   double xmin    =        x[0];
180   double xmax    =        x[0];

........ <<< Lines 181-209 omitted >>> ..........

210   gsSigma    = sigma;
211   gsNormalization = sqrt( twoPi ) * sigma * ( ymax - ymin );
212
213   // Setting up the axis system ................
214   double delta[2], minmax[4];
215
216   //  ______________ x       ___________y
217    minmax[0]  =  xmin;  minmax[2] = ymin;
218    minmax[1]  =  xmax;  minmax[3] = ymax;
219     delta[0]  =          delta[1] =  1.0;
220   fit.makeAxes( minmax, delta );
221                        minmax[2] = - 1.5;
222                        minmax[3] =   1.5;
223                         delta[1] =  0.25;
224   resid.makeAxes( minmax, delta );
225
226
227   // Plotting the data
228   dataGraf = fit.startPts( x[0], y[0], YELLOW );
229   for ( i = 1; i < num; i++ ) fit.addToPts( x[i], y[i] );
230
231   // Plot the initial curve
232   curve     =   fit.startFrame( x[0], gaussian( x[0] ), RED    );
233   residuals = resid.startFrame( x[0], 0.0,              YELLOW );
234               resid.addToFrame( x[0], y[0] - gaussian( x[0] )  );
235   for ( i = 1; i < num; i++ ) {
236     fit.addToFrame    ( x[i], gaussian( x[i] )         );
237     resid.joinToFrame ( x[i], 0.0                      );
238     resid.addToFrame  ( x[i], y[i] - gaussian( x[i] )  );
```

```
239     }
240   fit.purgeBuff();
241
242   // Activate the menu .....
243   ttMenu mainMenu;
244
245   mainMenu.setPrompt( "PLAYFIT ..> " );
246   mainMenu.setItem( "Absolute change of parameters.", absChange );
247   mainMenu.setItem( "Relative change of parameters.", relChange );
248
249   mainMenu.go();
250
251   }
```

2.3 Fields

In addition to wire frames and clusters, canvases recognize vector fields as representable objects. The program below samples an electric dipole field at random points.

lines 23-38: A canvas variable is declared and customized more or less as in Section 2.1 except that "orthographic" view is not requested, which means the display will be in perspective.

lines 45-52: The method **.startVF** initializes the vector field display. In the loop, **getFieldAtPoint** chooses a point randomly and calculates the field at that point. These are loaded respectively into the arrays `x` and `E`. Then, **.addToVF** (line 51) incorporates these new samples into the diagram as they are calculated.

PROGRAM LISTING

```
1     #include <stddef.h>
2     #include <errno.h>
3     #include <stdio.hxx>
4     #include <math.hxx>
5     #include "canvas.rsc"
6
7     extern double ran0( int* );
8
9     double       x[3], a[3], b[3], E[3];
10    int          j;
11    int          ranSeed = -9;
12    void         getFieldAtPoint();
13    inline void  getNextPoint() {
14      for( int k = 0; k < 3; k++ ) x[k] = - 8.0 + 16.0 * ( ran0( &ranSeed ) );
15      }
16
17    main() {
18
19    const int  samples = 2500;
20    int        i;
```

```
21    char*      dipoleField;
22
23    canvas3d   picture ( "OWW" );
24
25    // Put axes into the picture
26    double minmax[3];
27    double delta[3];
28    // ____ x  _________y  _________z
29    minmax[0] = minmax[2] = minmax[4] = - 5.0;
30    minmax[1] = minmax[3] = minmax[5] =   5.0;
31    delta [0] = delta [1] = delta [2] =   1.0;
32    picture.makeAxes( minmax, delta, RED );
33
34    // Connect transformations to dials ............
35    picture.connRot   ( 1, 2, 3 );
36    picture.connScale ( 4 );
37    picture.connFrom  ( 5 );
38    picture.connTrans ( 6, 7, 8 );
39
40    // Set positions of the two sources ............
41    //  ____ x   __________ y  ___________ z
42    a[0] = 0.0;    a[1] = 0.0;   a[2] =   0.5;
43    b[0] = 0.0;    b[1] = 0.0;   b[2] = - 0.5;
44
45    // Main loop ..................................
46    getFieldAtPoint();
47    dipoleField = picture.startVF( x, E );
48
49    for ( i = 0; i < samples; i++ ) {
50      getFieldAtPoint();
51      picture.addToVF( x, E );
52      }
53
54    picture.purgeBuff();
55    // Delaying program exit ......................
56    char  dummyString[ 80 ];
57    printf("\nPress RETURN to exit the program.\n");
58    gets( dummyString );
59
60    }
61
62    void getFieldAtPoint() {

..... <<< Lines 63-86 omitted >>> ..........

87    }
```

2.4 Fractals

A third flavor of canvas available for immediate use is **canvasRas**, a two-dimensional canvas designed for pixel operations: specifically, for scanning a region in R^2 and coloring pixels according to a user-specified algorithm. This is a useful technique for creating contour plots and is indispensable if one wants to become rich and famous, especially rich, by publishing fractal images.[6] The program shown below mixes a **canvas3d**, in which is plotted the Lorenz attractor, with three **canvasRas** variables, in which are displayed various portions of the Mandelbrot set.

lines 9-10: The external function **lorenz** propagates an orbit obeying Lorenz's differential equation, while **mandelbrot** determines the RGB color code of points in $C \simeq R^2$ based on the number of iterates of the complex function $f(z) = z^2 + c$ which "survive." We need not concern ourselves with their details here.

line 22: The last component of positioning arrays **rtf** and **lbb**, which appear in instancing the three-dimensional canvas **lorenzCnv**, specify positions for the forward and backward clipping planes. These are used in conjunction with depth cueing on the PS390: points which are "farther away" from the observer are made to appear dimmer. The first two components locate the canvas on the screen, as in lines 151 and 156 of Section 2.2.

lines 40-50: An orbit of the Lorenz equation is modelled as a wire frame beginning at the point $(0, 1, 0)$. Over five thousand time steps are taken. Since the transformation nodes have been connected prior to this, in lines 35-38, the user can manipulate the attractor while it is being constructed.

lines 81-87: Three regions of the Mandelbrot set will be displayed in raster canvases, whose declarations include arguments **rt** and **lb** indicating where they are to be placed on the screen, exactly as with **canvas2d** and **canvas3d** variables. The **.setX** and **.setY** functions then determine the domain in R^2 to be associated with each one, after which **.scan** tells the canvas to scan its domain using the RGB color-coding function named in the argument, **mandelbrot**.

PROGRAM LISTING

```
1     #include <stddef.h>
2     #include <errno.h>
3     #include <complex.hxx>
4
5     #include "canvas.rsc"
6
7     //  Declaring functions which generate Lorenz's attractor
8     //  and the Mandelbrot set.........................
9     extern void lorenz     ( double*, double*, double* );
10    extern void mandelbrot ( double*, double*, int*, int*, int* );
11
12    main() {
13
14    // ++++++++++++++++++++++++++++++++++++++++++++++++++++++
15    // Work begins here on the Lorenz attractor
16
```

[6] I am told that the current rate for a single fractal "project" in the advertising industry is about $150,000. At least, this is the price charged by a fractal pioneer who has left academia to devote himself full-time to these worthwhile pursuits.

```
17    // Declaring the canvas variable ..............
18    double  rtf[3], lbb[3];
19    // ___________x   _____________y    _____________z
20    lbb[0]  =  -1.0;  lbb[1]  = -0.6;   lbb[2]  = -1.0;
21    rtf[0]  =   0.6;  rtf[1]  =  1.0;   rtf[2]  =  1.0;
22    canvas3d lorenzCnv( rtf, lbb, "OWO" );
23
24    // Setting up the axis system .................
25    double  delta[3];
26    double minmax[6];
27    // ____ x    _________y   ________z
28    minmax[0]  =  minmax[2] = minmax[4] =  - 50.0;
29    minmax[1]  =  minmax[3] = minmax[5] =    50.0;
30     delta[0]  =   delta[1] =  delta[2] =     5.0;
31    lorenzCnv.makeAxes( minmax, delta, YELLOW );
32
33
34    // Connect transformations to dials ............
35    lorenzCnv.connRot   ( 1, 2, 3 );
36    lorenzCnv.connScale ( 4 );
37    lorenzCnv.connTrans ( 5, 6, 7 );
38    lorenzCnv.connFrom  ( 8 );
39
40    // Set initial conditions and launch the orbit .....
41    double x = 0.0;
42    double y = 1.0;
43    double z = 0.0;
44
45    char* attractor;
46    attractor = lorenzCnv.startFrame( x, y, z, RED );
47    for ( int timeSteps = 0; timeSteps < 5500; timeSteps++ ) {
48      lorenz( &x, &y, &z );
49      lorenzCnv.addToFrame( x, y, z );
50      }
51
52    // ++++++++++++++++++++++++++++++++++++++++++++++++++++++
53    // Work begins here on the Mandelbrot sets
54
55    // Declare five raster canvases in which we shall scan
56    // various pieces of the Mandelbrot set.
57
58    double rt[3][2];
59    double lb[3][2];
60    // ______________x   _________________y
61    lb[0][0]  =    0.6;  lb[0][1]  = - 0.2;
62    rt[0][0]  =    1.0;  rt[0][1]  =   0.2;
63    // ----------------  ------------------
```

```
64    lb[1][0]  =    0.6;  lb[1][1]  = - 1.0;
65    rt[1][0]  =    1.0;  rt[1][1]  = - 0.6;
66    // ---------------  -----------------
67    lb[2][0]  =  - 0.2;  lb[2][1]  = - 1.0;
68    rt[2][0]  =    0.2;  rt[2][1]  = - 0.6;
69
70    double xLo[3], xHi[3], yLo[3], yHi[3];
71    xLo[0] = -1.7935;    xHi[0] = -1.7365;
72    yLo[0] = -0.0285;    yHi[0] =  0.0285;
73    // ---------------  -----------------
74    xLo[1] = -0.96;      xHi[1] = -0.89;
75    yLo[1] =  0.23;      yHi[1] =  0.30;
76    // ---------------  -----------------
77    xLo[2] = -0.94;      xHi[2] = -0.89;
78    yLo[2] =  0.24;      yHi[2] =  0.29;
79    // ---------------  -----------------
80
81    canvasRas* mandelCnv;
82    for ( int i = 0; i < 3 i++ ) {
83      mandelCnv =  new canvasRas(  &( rt[i][0] ), &( lb[i][0] )  );
84      mandelCnv -> setX( xLo[i], xHi[i] );
85      mandelCnv -> setY( yLo[i], yHi[i] );
86      mandelCnv -> scan( mandelbrot );
87      }
88
89    // Delaying program exit ......................
90    char  dummyString[ 80 ];
91    printf("\nPress RETURN to exit the program.\n");
92    gets( dummyString );
93
94    }
```

3 Object-oriented graphics

In order to reduce the amount of space needed to list methods available to canvas variables, we have adopted a limited form for indicating options. Using this notation the line,

{ classA | classB }::mth(double x[3], [int* y ,] char* z [, double t [, int w]])

means that the method **.mth** is available both to **classA** and to **classB** using one of the following argument lists:

(double x[3], char* z),
(double x[3], int* y, char* z, double t), or
(double x[3], int* y, char* z, double t, int w).

However, (**double x[3], int* y, char* z**) would *not* be allowed: unless otherwise indicated, all options at the same depth go together.

3.1 Declarations

canvas3d::canvas3d ([char* m [, char* v]])
::canvas3d (double rtf[3], double lbb[3] , char* m [, char* v])
canvas2d::canvas2d ([[double rt[2], double lb[2],] char* m])

Canvas constructors are overloaded to provide various options for initializing the graphics display environment. The argument (**char* m**) requires some explanation. Three types of "real-time," interactive transformations are incorporated into canvases: rotations, dilations, and translations.[7] The argument `m` is a three-character string (excluding the terminating null character) which specifies the *mode* in which these transformations are to be carried out. For instance, rotations are carried out relative to three independent directions. If the first character in `m` is 'W', this means that these directions are to remain fixed in space (or, in graphics terminology, in the "world"); if it is 'O' then the rotation axes remain fixed relative to the objects being rotated. In the latter case, dials connected to the rotation node of a canvas (via one of the **.connRot** methods discussed below) will act like yaw, pitch, and roll controls. If 'W' is specified, however, the axes of rotation will remain fixed "in space" regardless of the orientation of the objects.

A similar criterion is set up with regard to translations, which also require specification of three axes, using the third character in `m`. The middle character controls the dilation, or "scaling," operation: if 'O' is chosen, then the invariant point of the dilations remains fixed in the object; if 'W', in the world.

The argument **char* v**, if present, gives the user the option of specifying an orthographic view, rather than a perspective one: Any string not beginning with the character 'p' — such as `v = "orthographic"` — will select the orthographic view. This will perform parallel projection of three-dimensional data onto the screen.

If more than one canvas is declared, the arrays **rt**, **lb**, **rtf**, and **lbb** indicate the size and position of each on the screen. The two components of **rt** and **lb** contain the *relative screen coordinates* of the right-top and left-bottom vertices of the canvas. Relative screen coordinates are always in the range ± 1 in each direction: they indicate position as though the screen were mapped onto $\{ (x, y) \mid |x| \leq 1,\ |y| \leq 1 \}$. For example, using $\mathrm{rt} = (1, 1)$ and $\mathrm{lb} = (0, 0)$ will put the viewport in the upper right hand portion of the screen. If a rectangular viewport is requested, the one actually constructed will enclose the *largest* square that fits into this rectangle, with top right vertices matching. The three-dimensional arrays **rtf** and **lbb** are similar; their third argument indicates "depth," and sets the positions of the forward and backward clipping planes, which determine depth cueing.

Default declarations use no arguments. A canvas constructor assumes that the declared canvas will be the only one on the screen and give it maximum size. Data will appear in perspective, and depth cueing will be activated with forward and backward clipping planes set at ± 1.

[7] There is actually one more, but we'll save it for later.

3.2 Transformations

```
void canvas3d::connRot ( int dialX, [ char* labelX, ]
                         int dialY, [ char* labelY, ]
                         int dialZ  [ , char* labelZ ] )
               ::connTrans ( int dialX, [ char* labelX, ]
                             int dialY, [ char* labelY, ]
                             int dialZ  [ , char* labelZ ] )
               ::connFrom ( int dial [ , char* label ] )
void canvas2d::connRot   ( int dial [ , char* label ] )
               ::connTrans ( int dialX, [ char* labelX, ]
                             int dialY [ , char* labelY ] )
void { canvas3d | canvas2d }::connScale( int dial [ , char* label ] )
```

These functions connect the transformation nodes of a canvas to the PS390's dial box so that they can be carried out interactively by turning knobs. As their names suggest, **.connRot** and **.connTrans** connect dials to a canvas's rotation and translation nodes respectively, and **.connScale** connects a dial to dilation. The integer arguments determine the dials associated with "x," "y," and "z" rotations – or just "z" rotations, in the two-dimensional case – and must lie between 1 and 8, inclusively; the string arguments determine the labels, limited to eight characters, which will appear in the LED displays above the associated dials. If labels are not specified, defaults will be used. An input dial integer less than 1 or larger than 8 will result in no transformation capability about the axis for which the dial number was intended. This is actually useful when it is desired to select fewer than all three axes for either rotations or translations.

.connScale requires only one integer argument for selecting the dial number to be used for dilation, as the transformation is carried out uniformly in all three dimensions. It is important to remember that the origin used for the scaling tranformation is determined by the mode argument, `char* m`, used when declaring a variable of type **canvas3d**.

The **.connFrom** method, available only to **canvas3d** variables, allow the user to "move through" the data by changing the point of observation. Any rotation of the selected dial will result in the simulated movement of the user along the line connecting the original viewing position and the world axis system origin. If the viewing position is moved beyond the origin (the center of the display region), the data will apear to flip from front to back.[8]

3.3 Building displayable objects

```
char* canvas3d::startFrame( double x, double y, double z [ , double c ] )
                ::startPts   ( double x, double y, double z [ , double c ] )
                ::startVF    ( [ double s, ] double p[3], double v[3] [ , double c ] )
void canvas3d  ::addToFrame ( [ char* n, ] double x, double y, double z )
                ::joinToFrame ( [ char* n, ] double x, double y, double z )
                ::addToPts    ( [ char* n, ] double x, double y, double z )
```

[8] This is a flaw which will be repaired eventually.

```
                ::addToVF     ( [ char* n, ] double p[3], double v[3] )
char* canvas2d::startFrame ( double x, double y [ , double c ] )
                ::startPts    ( double x, double y [ , double c ] )
                ::startVF     ( [ double s, ] double p[2], double v[2] [ , double c ] )
void canvas2d  ::addToFrame ( [ char* n, ] double x, double y )
                ::joinToFrame ( [ char* n, ] double x, double y )
                ::addToPts    ( [ char* n, ] double x, double y )
                ::addToVF     ( [ char* n, ] double p[2], double v[2] )
void { canvas3d | canvas2d }::makeAxes( double m[6], double d[3] [ , double c ] )
```

At the time of writing, canvases support displayable objects modelled as wireframes, point sets, vector fields, or axes. Displaying data as clusters of points is accomplished through the functions **.startPts** and **.addToPts**. The first initializes the object, and the second adds members to it. **.startFrame**, **.addToFrame**, and **.joinToFrame** perform similar offices for wireframes. **.addToFrame** assumes the point given is to be attached to the frame via an edge; **.joinToFrame** makes the edge invisible: it "lifts the pen," as it were, before moving to the new point. Thus, a wire frame need not appear connected.

The **.startXXX** methods return a string which is the internal name of the object being created. One application of this, the removal of an object from a canvas, was illustrated in line 51 of the Section 2.2 listing. Another is given in the methods above: Normally, after an object is **started**, calls are made to **.addToXXX** methods in order to continue its construction, as done in lines 231-239 of Section 2.2. If that process is interrupted, and *another* object is **started**, we can still return to the original one by inserting (the token containing) the value returned by a **.startXXX** function as the **char* n** argument. This is illustrated in the fragment below.

```
canvas3d  zlorfik;
char*     firstThing;
char*     secondThing;
double    x, y, z;
...
firstThing  = zlorfik.startFrame( 0., 0., 0. );
secondThing = zlorfik.startFrame( 1., 1., 1. );
zlorfik.addToFrame( x, y, z );  // This refers automatically to
                                // the most recently started frame
zlorfik.addToFrame( firstThing, x, y, z );
zlorfik.joinToFrame( secondThing, 2.0*x, 2.0*y, 2.0*z );
```

Axes, built by the **.makeAxes** functions, are useful for orienting the observer with respect to data. The two required arguments are arrays which provide the ranges associated with each axis (**double* m**) and the spacing between tic marks (**double* d**). Their usage was illustrated in all Section 2, for example in lines 16-19 of Section 2.1. Of course, the last (two) components of **double* d** (**double* m**) is (are) ignored by, and therefore not necessary for, **canvas2d** variables.

The optional argument **double s** that appears in the vector field function **.startVF** is a multiplicative

"scale factor" for the vectors which serves to relate the units of the vector field to those of the underlying space. Its absence is equivalent to setting `s = 1`.

The argument **double c** refers to color. **BLUE**, **MAGENTA**, **RED**, **YELLOW**, **GREEN** and **CYAN** are automatically available as defined tokens, and can be inserted into this slot. Other hues can be obtained through interpolation, as in "`0.1*GREEN + 0.9*YELLOW`."

3.4 Miscellany

```
void canvas3d::setRange( double xMin, double xMax,
                         double yMin, double yMax,
                         double zMin, double zMax )
void canvas2d::setRange( double xMin, double xMax,
                         double yMin, double yMax )
void { canvas3d | canvas2d }::setRange( double m[6] )
                             ::include( char* n )
                             ::remove( char* n )
void canvas3d::connAnimate( char g, int s )
void canvas2d::connAnimate( int s )
```

Besides arranging the configuration of the axes, the **.makeAxes** methods listed in Section 3.3 alert the canvas to the range of values it can expect. If necessary, the canvas then scales data so that all directions visually look *as though they had comparable dynamic ranges*. The **.setRange** methods provide a way of doing this *without* drawing axes. The argument **double* m** is exactly the same as in Section 3.3; as before, the last two components are not needed for **canvas2d** variables.

After an object is created in one canvas, it can be displayed in other canvases by using the **.include** function. Conversely, **.remove** will remove objects from canvases, as was done in lines 51-52 of Section 2.2.

.connAnimate allows the user to activate an animated rotation about an axis of choice, determined by the argument **char g**, which should take on the values `'x'`, `'y'`, or `'z'`. The other argument, **int s**, controls the rotation speed: a value of 1 causes the animation to operate at a speed of about 60 degrees per second, which is the fastest available; any other value will produce a slower rotation. The animation is begun by placing the locator puck on the tablet and pressing its green button, and it is stopped by pressing the red one.

4 Raster Graphics

The class **canvasRas**, unlike its cousins, **canvas2d** and **canvas3d**, is designed for pixel operations. Specifically, it can be used to generate color-coded contour maps of functions over R^2. As was demonstrated in Section 2.4, the user specifies an arbitrarily named function

```
void func( double* x, double* y, double* r, double* g, double* b )
```

which accepts an (x, y) coordinate pair as input and produces its red-green-blue color code as output. The values must all satisfy $0 \leq$ `*r, *g, *b` < 255. Pixel operations *are not updated,* so **canvasRas** variables should not overlap anything else appearing on the screen, including the VAX-produced messages displayed in the terminal emulator plane and the cursor.

4.1 Declarations

canvasRas::canvasRas([double rt[2], double lb[2]])

Instancing a **canvasRas** variable with arguments determines its position on the screen. (For example, see line 83 on page 14.) As before, `rt` and `lb` contain normalized screen coordinates, lying between ± 1. If the default option (no arguments) is used, then the canvas will fill the entire available area, which means that it should be the *only* canvas in the program.

4.2 Scanning the function

canvasRas::setX(double lo, double hi)
::setY(double lo, double hi)
::scan(RASTER_FUNC scanFunction)

The desired image is rendered on the screen by setting a domain in R^2 and then invoking **.scan**. The methods **.setX** and **.setY** determine the region to be scanned. Invoking **.scan** starts the actual data processing. The argument `scanFunction` is the name of the user-defined function described above. Usage was exemplified in lines 10 and 86 of Sec.2.4.

4.3 Miscellany

canvasRas::erase()
::fullErase()

Invoking the **.erase** method will clear a single raster canvas; invoking **.fullErase** will clear *all* raster canvases on the screen.